# 深度数据包检测技术研究进展

黄昆，谢高岗

**摘要** 深度数据包检测是数据包处理关键技术之一，即采用特征匹配算法，将每个数据包内容与一组预定义的特征进行匹配。随着网络带宽的迅猛增长以及特征规则日益增多，研究者提出了基于硬件的特征匹配算法，即采用 FPGA、ASIC 和 NP 等专用嵌入式硬件来设计与实现特征匹配算法，提高 DPI 的匹配吞吐量。但是，这些基于硬件的特征匹配算法面临高性能挑战，即如何满足线速数据包内容过滤的时间和空间需求。本文从时间和空间等方面综述了基于硬件的字符串匹配算法和正则表达式匹配算法的研究进展，并展望了未来 DPI 技术研究。

**关键词** 网络安全 深度数据包检测 特征匹配 字符串匹配 正则表达式



## 1 引言

随着网络技术的迅猛发展和广泛应用，互联网承载着越来越多的应用业务，人们对互联网的依赖性日益增强。近年来，网络蠕虫、僵尸网络和计算机病毒等新型攻击层出不穷，入侵和劫持大量的计算机系统，滥用计算机网络资源，并威胁和破坏互联网基础设施，已经造成重大的经济损失和恶劣的社会影响[1-2]。因此，如何快速检测和阻止这些攻击成为当前网络安全研究的热点问题。

网络入侵检测与防御系统(Network Intrusion Detection/Prevention System, NIDS/NIPS)是网络安全防御的主要手段，即通过实时监测网络流量来检查和阻断网络攻击[1]。NIDS/NIPS 已广泛部署于从终端计算机、边缘路由器到核心路由器等网络构件。随着网络攻击日益复杂化和安全漏洞不断发现，在未来五年内，全球 NIDS/NIPS 市场将从 2007 年的 9.32 亿美元增长到 21 亿美元[3]。深度数据包检测(Deep Packet Inspection, DPI)是 NIDS/NIPS 的核心，不仅检查数据包头部信息，而且检查数据包有效载荷(即数据包内容)。深度数据包检测（以下简称 DPI）主要采用特征匹配算法，即将每个数据包内容与一组预定义的特征规则进行匹配[4-5]。DPI 采用一组规则描述攻击特征，即每条规则至少包括数据包类型、特征字符串、搜索起始位置、以及匹配后的响应操作等。例如，在 Snort 特征规则集中[6]，规则{alert icmp (msg: "DDOS TFN Probe"; content: "1234";)}是描述分布式拒绝服务 TFN[1]探测攻击，即当互联网控制报文协议（Internet Control Message Protocol, ICMP）数据包内容包含特征字符串“1234”时，产生告警消息为“DDOS TFN Probe”。DPI 是一种数据包内容过滤技术，不仅应用于 NIDS/NIPS，而且还应用于应用层数据包分类、对等传输（P2P）流量识别、基于上下文的流量计费等[7-8]。

特征匹配是计算机科学中经典问题之一，广泛应用于信息检索、模式识别和网络安全等领域。特征匹配算法可分为字符串匹配算法和正则表达式匹配算法。字符串匹配算法采用字符串语言来描述简单的特征规则；而正则表达式匹配算法采用正则表达式语言来描述复杂的特征规则。根据匹配方式，特征匹配算法又分为单模式和多模式匹配算法。单模式匹配算法是指一次内容扫描仅匹配一个特征串，例如克努斯-莫里斯-普拉特（Knuth-Morris-Pratt）算法[9]和博耶-摩尔（Boyer-Moore）算法[10]等。当规则集包含 $s$ 个特征串时，单模式匹配算法

[1] Tribe Flood Network

需要重复匹配 *s* 次，存在匹配效率低等问题。多模式匹配算法是采用确定型有限自动机(Deterministic Finite Automaton，DFA)来表示一组特征串，一次内容扫描可匹配多个特征串，例如阿霍-克拉斯科（Aho-Corasick）算法[11]、科曼兹-沃尔特（Commentz-Walter）算法[12]以及阿霍-克拉斯科和博耶-摩尔（Aho-Corasick Boyer-Moore）联合算法[13]等。

从 1975 年开始，过去近 30 多年的 DPI 技术研究主要关注基于软件的特征匹配算法，即在单核 CPU 的软件平台上设计与实现特征匹配的算法，从而提高 DPI 的匹配速率。但是，随着网络带宽和业务流量的迅猛增长以及特征规则日益增多，基于软件的特征匹配算法存在匹配吞吐量低等问题，难以满足 10-40Gbps 线速数据包处理的高性能需求。近年来，研究者提出了多种基于硬件的特征匹配算法[14-18]，即采用专用嵌入式硬件，例如现场可编程门阵列（简称门阵列）、专用集成电路等，设计与实现特征匹配算法，从而提高 DPI 的匹配吞吐量。例如，基于门阵列的特征匹配算法吞吐量可达 10Gbps，支持特征规则集的动态更新，但是存在重编译和重构时间长等缺点；基于专用集成电路的特征匹配算法吞吐量可达 20Gbps，不支持特征规则集的动态更新。这些基于硬件的特征匹配算法面临嵌入式存储器空间受限等挑战，即确定型有限自动机（以下简称确定自动机）存储空间需求大，难以在片上高速存储器中存储和查找整个确定自动机，而需要频繁访问片外低速存储器，导致 DPI 的匹配性能降低。例如，Xilinx Vertex-5 门阵列[19]仅提供约 10Mb 的片上 SRAM[2]，而当前 Snort 特征规则集对应的确定自动机存储空间开销约为 100MB，无法存储在片上 SRAM 中，难以实现线速数据包处理。TCAM[3]和多核处理器(Multi-Core Processor)等硬件技术的不断发展，为 DPI 的性能提升提供了新的机遇与挑战。TCAM 可以在一个时钟周期内查找出与内容匹配的存储索引，提供高速且确定的内容查找，用于加速 DPI 的匹配性能；多核处理器是在一块芯片上集成多个 CPU 核，具有高速灵活的并行计算能力，用于提升 DPI 的匹配性能。因此，如何设计与实现基于硬件的高性能特征匹配算法成为线速 DPI 技术研究的热点问题，已吸引越来越多研究者的关注。

本文综述了 DPI 技术研究进展，特别是基于硬件的特征匹配算法设计与实现。第 2 节介绍了 DPI 的高性能挑战，并指出了 DPI 技术的可伸缩性需求；第 3 节和第 4 节分别详细讨论了几种基于硬件的字符串匹配算法和正则表达式匹配算法。最后，第 5 节总结全文，并展望了未来 DPI 技术研究的关键问题。



# 2 DPI 技术的高性能挑战

DPI 是计算密集型操作，主要应用于高速路由器的关键数据路径上，需要检查高速海量数据包，并与成千上万条规则进行特征匹配。近年来，互联网骨干链路带宽从 2.5Gbps 增至 10-40Gbps，以太网接口从 10GbE 增至 100GbE；新型高质量网络应用，例如流媒体应用 PPLive、YouTube，内容发布应用 Facebook、Twitter 等，产生海量的业务内容和网络流量。为了适应高速海量数据包处理，满足线速数据包处理的时间和空间需求，并提升可伸缩性，DPI 技术面临高性能挑战[20]：

1. 随着网络攻击、业务行为等日益复杂化，特征规则条数不断增多且特征描述越来越复杂，导致 DPI 的存储空间开销日益增大。例如，Snort 特征规则条数从 2003 年的 3166 增至 2009 年的 15047。Snort 特征规则集的确定自动机存储空间需求超过 75MB，而门阵列和专用集成电路等嵌入式硬件的片上高速存储器仅约为 10Mb，因而特征匹配的整个确定自动机难以存储在片上高速存储器中，需要存储在片外低速存储器，

[2] Static Random Access Memory，静态随机存储器
[3] Ternary Content Addressable Memory, 三态内容可寻址存储器

从而限制了 DPI 的匹配吞吐量。因此，基于硬件的 DPI 技术迫切要求存储高效特征匹配算法，在片上高速存储器中存储和查找确定自动机，不仅满足特征规则集的可伸缩性需求，而且提高 DPI 的匹配性能。

2. 为了实现未来 100Gbps 线速数据包处理，基于硬件的 DPI 技术要求利用片上高速存储器、硬件并行计算能力等来提高特征匹配的吞吐量。门阵列和专用集成电路等嵌入式硬件通常采用层次化存储器体系结构，即由片上高速存储器和片外低速存储器构成。片上存储器支持快速查找，例如片上 SRAM 的访问时间为 1-2ns，但是其存储空间小；片外存储器的存储空间大，但是其查找速度慢，例如片外 DRAM[4]的访问时间为 60ns。面向硬件实现的 DPI 技术面临如何利用片上高速存储器来尽量减少片外低速存储器访问次数，从而实现线速 DPI。专用嵌入式硬件虽然支持并行处理，但是存在编程设计复杂、灵活性差和升级成本高等缺点。Intel Xeon CPU、Cavium OCTEON 和 Nivida GPGPU 等多核处理器具有强大并行计算能力和灵活可编程等优点，为线速 DPI 实现带来新的机遇与挑战。线速 DPI 技术面临如何利用多核处理器平台的并行处理能力，设计与实现时空高效特征匹配算法，从而加速 DPI 的匹配性能的挑战。

总之，DPI 的可伸缩性需求主要体现在存储空间开销和匹配吞吐量等两方面[21]，即减少特征匹配算法的存储空间开销，并提高其匹配速率，从而实现高性能 DPI。因此，当前的 DPI 技术研究主要是从时间和空间等方面来设计与实现基于硬件的特征匹配算法，从而满足 DPI 的高性能需求。本文是以时空开销为视角，探讨基于硬件的字符串匹配算法和正则表达式算法等研究进展，并展望未来 DPI 技术研究的关键问题。

## 3 基于硬件的字符串匹配算法

基于硬件的字符串匹配算法可分为：存储高效字符串匹配算法、多字符字符串匹配算法和并行字符串匹配算法。如表 1 所示，存储高效算法是研究如何通过减少冗余迁移边来压缩确定自动机存储空间开销，例如基于 B-FSM[5][22]和基于 CDFA[6][23]的阿霍-克拉斯科算法；多

**表 1** 基于硬件的字符串匹配算法分类

| | 存储高效算法 | 多字符算法 | 并行算法 |
|---|---|---|---|
| B-FSM | √ | | |
| CDFA | √ | | |
| JACK-NFA | | √ | |
| TDP-DFA | | √ | |
| 压缩 DFA | | √ | |
| 变步长 DFA | | √ | |
| 比特拆分 DFA | | | √ |
| 首尾分割 DFA | | | √ |
| 并行布隆过滤器 | | | √ |

字符算法是研究如何通过构建一次处理多个字符的确定自动机来提高其匹配吞吐量，例如基于 JACK-NFA[7][24]、基于 TDP-DFA[8][25]、基于压缩的确定型有限自动机[26]和基于变步长的确定型有限自动机[27]的阿霍-克拉斯科算法；并行算法是研究如何通过并行化字符串匹配操作

[4] Dynamic Random Access Memory，动态随机存储器
[5] BART-based Finite State Machine, BART：Balanced Routing Table，一种可编程状态机技术
[6] Cached Deterministic Finite Automaton，具有缓存的确定型有限自动机
[7] Jump-ahead Aho-CorasicK NFA，NFA：Non-deterministic Finite Automaton，非确定型有限自动机
[8] Transition-Distributed Parallel DFA，迁移边分布的并行确定型有限自动机

来加速确定自动机的匹配性能，例如基于比特拆分的确定自动机[28]、基于首尾分割的确定自动机[29]和基于并行布隆过滤器（Bloom Filter）[30]的阿霍-克拉斯科算法。

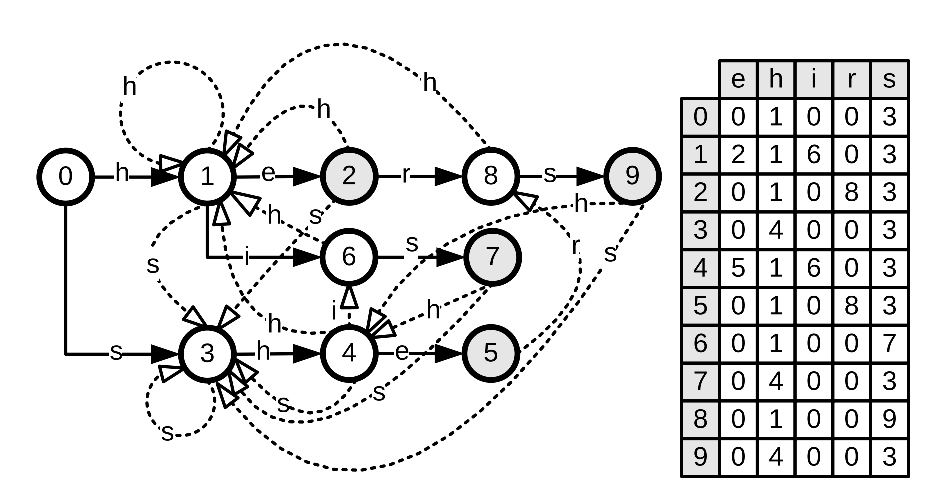
## 3.1 阿霍-克拉斯科算法

阿霍-克拉斯科算法[11]是经典的字符串匹配算法，即采用一个确定自动机表示一组特征字符串，称为原始确定自动机。原始确定自动机的构建过程是：

- 为特征字符串集构建一颗以初始状态 0 为根节点的特里树(Trie)；
- 从特征字符串集中提取所有唯一字符，构建字母表；
- 根据字母表，从特里树的根节点(即初始状态 0 )开始，逐层为每个状态构建其转移(Goto)函数、失效(Failure)函数和输出(Output)函数，直到叶子节点(即匹配状态)。

转移函数是指当读入字符匹配时，从当前状态迁移到下一个状态；失效函数是指当读入字符不匹配时，从当前状态迁移到某个指定状态；输出函数是指当迁移到匹配状态时，输出所有匹配字符串。

图 1 给出了字符串集 $\{\boldsymbol{he},\boldsymbol{she},\boldsymbol{his},\boldsymbol{hers}\}$ 的原始确定自动机，其中初始状态为 0 ，匹配状态为 2 、5 、7 和 9 。如图 1 所示，实线表示基本迁移边，即从树深度为 $\boldsymbol{d}$ 的源状态 $\boldsymbol{i}$ 迁移到树深度为 $\boldsymbol{d}+1$ 的目的状态 $\boldsymbol{j}$ ；虚线表示交叉迁移边，即从树深度为 $\boldsymbol{d}$ 的源状态 $\boldsymbol{i}$ 迁移到树深度为 $\boldsymbol{d'} \le \boldsymbol{d}$ 的目的状态 $\boldsymbol{j}$ ；如果目的状态 $\boldsymbol{j}$ 的树深度 $\boldsymbol{d'}$ 为 0，则该交叉迁移边又称为失效迁移边；如果目的状态 $\boldsymbol{j}$ 的树深度 $\boldsymbol{d'}$ 为 1，则该交叉迁移边又称为重启迁移边。如图 1 所示，在原始确定自动机的状态迁移表中，行首表示源状态，其他表示目的状态；列首表示匹配字符。注意：图 1 省略了原始确定自动机的失效迁移边。



| | e | h | i | r | s |
|---|---|---|---|---|---|
| 0 | 0 | 1 | 0 | 0 | 3 |
| 1 | 2 | 1 | 6 | 0 | 3 |
| 2 | 0 | 1 | 0 | 8 | 3 |
| 3 | 0 | 4 | 0 | 0 | 3 |
| 4 | 5 | 1 | 6 | 0 | 3 |
| 5 | 0 | 1 | 0 | 8 | 3 |
| 6 | 0 | 1 | 0 | 0 | 7 |
| 7 | 0 | 4 | 0 | 0 | 3 |
| 8 | 0 | 1 | 0 | 0 | 9 |
| 9 | 0 | 4 | 0 | 0 | 3 |

图1. 字符串集 $\{\boldsymbol{he},\boldsymbol{she},\boldsymbol{his},\boldsymbol{hers}\}$ 的原始确定自动机

原始确定自动机的匹配过程是：设置初始当前状态为 0 ，当读入一个字符时，当前状态迁移到下一个状态，设置下一个状态为当前状态，并依次状态迁移；如果下一个状态为匹配状态，输出所有匹配字符串。例如，在图 1 中，当读入一个字符串 $\{\boldsymbol{sohershe}\}$ 时，从初始状态 0 开始，原始确定自动机执行状态迁移为 $0 \to 3 \to 0 \to 1 \to 2 \to 8 \to 9 \to 4 \to 5$ ，并输出匹配字符串集

塔克（N. Tuck）等人[31]采用位图压缩和路径压缩等方法，减少阿霍-克拉斯科算法的确定自动机存储空间需求，并增强阿霍-克拉斯科算法的最坏情况性能。由于 ASCII 码字母表大小为 256，原始确定自动机的每个状态包含 256 个状态指针，指向每个 ASCII 码字符对应

的下一状态，导致原始确定自动机的存储空间开销很大。如图 2 所示，位图压缩确定自动机采用 1 个状态指针 ***Next Ptr*** 及其节点位图 ***Node Bitmap***、1 个失效指针 ***Failure Ptr*** 以及 1 个规则指针 ***Rule Ptr***，而不需要维护 256 个状态指针。状态指针 ***Next Ptr*** 是指向下一状态队列的头指针；节点位图 ***Node Bitmap*** 是表示每个 ASCII 码字符对应的下一状态是否是初始状态，即 0 比特表示下一状态是初始状态，1 比特表示下一状态不是初始状态；失效指针 ***Failure Ptr*** 是指向初始状态的指针；规则指针 ***Rule Ptr*** 是指向匹配规则队列的头指针。图 2 给出了位图压缩确定自动机的状态 5 数据结构。在 ***Node Bitmap*** 中，字符 ***e*** 和 ***i*** 对应的比特为 0，而字符 ***h***、***r*** 和 ***s*** 对应的比特为 1，因而 ***Next Ptr*** 指向下一状点队列的头节点 1，***Failure Ptr*** 指向初始状态 0，***Rule Ptr*** 指向匹配规则队列 {***she***,***he***}。

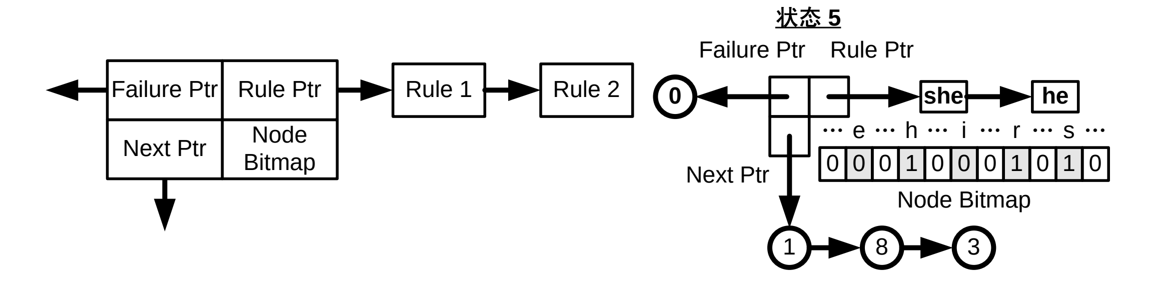



图2. 位图压缩确定自动机的状态节点数据结构

在图 2 中，当读入一个字符 ***r*** 时，由于节点位图 ***Node Bitmap*** 的第 ***r*** 位比特为 1，状态 5 计算出 ***Node Bitmap*** 中第 ***r*** 位之前的 1 比特个数为 1，则计算出下一状态的存储地址是 ***Next Ptr*** +1，并检查该状态的规则指针，输出对应的匹配规则。当读入一个字符 ***i*** 时，由于节点位图 ***Node Bitmap*** 的第 ***i*** 位比特为 1，则计算出下一状态的存储地址是失效指针 ***Failure Ptr*** 指向的初始状态。

## 3.2 存储高效字符串匹配算法

确定自动机存储空间是由状态迁移边所构成。研究者提出了存储高效字符串匹配算法，通过减少状态之间的冗余迁移边来减少确定自动机存储空间需求。本节主要介绍基于 B-FSM 和基于 CDFA 的阿霍-克拉斯科算法。

### 3.2.1 基于 B-FSM 的阿霍-克拉斯科算法

伦特任(J. van Lunteren)等人[22]提出了一种基于 B-FSM(BART-based Finite State Machine)的阿霍-克拉斯科算法，即采用优先级和通配符状态等方法压缩状态迁移边，并采用 BART(Balanced Routing Table, 平衡路径表)算法搜索优先级最高的匹配迁移边。图 3 给出了字符串集 {***he***,***she***,***his***,***hers***} 的 B-FSM，其中左图是状态迁移，右图是优先级分别为 2、1、和 0 的迁移边。如图 3 所示，每条迁移边包含 4 个字段，即源状态、匹配字符、目的状态和优先级。例如，迁移边 $\{\boldsymbol{e}: 1 \rightarrow 2: 2\}$ 表示当匹配字符 ***e*** 时，源状态 1 迁移到目的状态 2 且优先级为 2 迁移边 $\{\boldsymbol{h}: * \rightarrow 1: 1\}$ 表示当匹配字符 ***h*** 时，任意状态迁移到目的状态 1 且优先级为 1；迁移边 $\{*: * \rightarrow 0: 0\}$ 表示当匹配非字母表字符时，任意状态迁移到目的状态 0，且优先级为 0。

B-FSM 的匹配过程是：当读入一个字符时，从状态迁移表中查找优先级最高的与当前状态和读入字符匹配的迁移边，即从优先级为 2 的迁移边开始查找，直到优先级为 0 的迁移边。例如，在图 3 中，当读入一个字符串 {***sohershe***} 时，在 B-FSM 中查找所有匹配字符串；当读入一个字符 ***s*** 时，当前状态为初始状态 0，匹配出迁移边 $\{\boldsymbol{s}: * \rightarrow 3: 1\}$，迁移到下一状

态 3；当读入下一个字符 ***o*** 时，匹配出迁移边 $\{*:*\rightarrow 0:0\}$，迁移到下一状态 0；当读入下一个字符 ***h*** 时，匹配出迁移边 $\{\boldsymbol{h}:*\rightarrow 1:1\}$，迁移到下一状态 1；当读入下一个字符 ***e*** 时，匹配出迁移边 $\{\boldsymbol{e}:1\rightarrow 2:2\}$，迁移到下一状态 2，输出匹配字符串 $\{\boldsymbol{he}\}$；当读入下一个字符 ***r*** 时，匹配出迁移边 $\{\boldsymbol{r}:2\rightarrow 8:2\}$，迁移到下一状态 8；当读入下一个字符 ***s*** 时，匹配出迁移边 $\{\boldsymbol{s}:8\rightarrow 9:2\}$，迁移到下一状态 9，输出匹配字符串 $\{\boldsymbol{hers}\}$；当读入下一个字符 ***h*** 时，匹配出迁移边 $\{\boldsymbol{h}:9\rightarrow 4:2\}$，迁移到下一状态 4；当读入下一个字符 ***e*** 时，匹配出迁移边 $\{\boldsymbol{e}:4\rightarrow 5:2\}$，迁移到下一状态 5，输出匹配字符串 $\{\boldsymbol{she},\boldsymbol{he}\}$。因此，B-FSM 执行状态迁移为 $\{0\rightarrow 3\rightarrow 0\rightarrow 1\rightarrow 2\rightarrow 8\rightarrow 9\rightarrow 4\rightarrow 5\}$，并输出匹配字符串集 $\{\boldsymbol{he},\boldsymbol{hers},\boldsymbol{she},\boldsymbol{he}\}$。

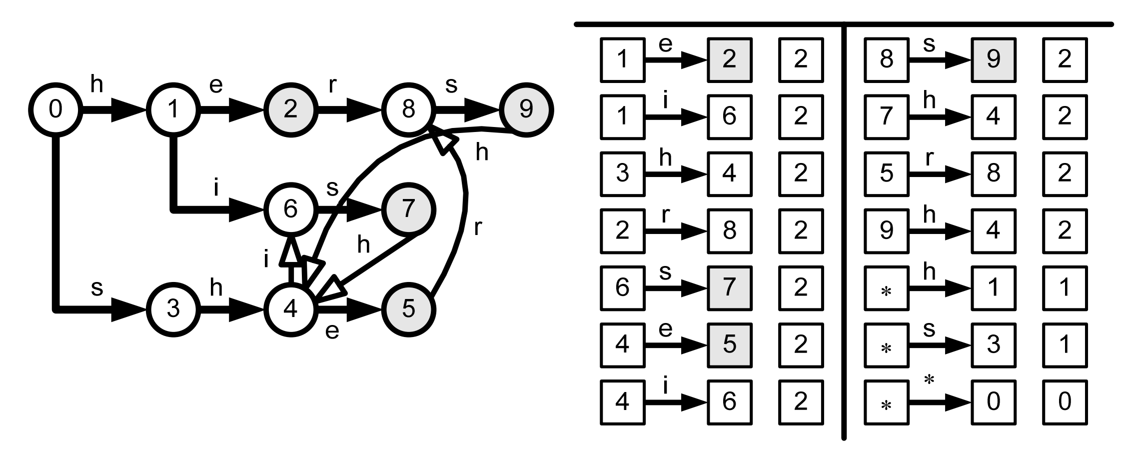



图3. 字符串集 $\{\boldsymbol{he},\boldsymbol{she},\boldsymbol{his},\boldsymbol{hers}\}$ 的 B-FSM

### 3.2.2 基于 CDFA 的阿霍-克拉斯科算法

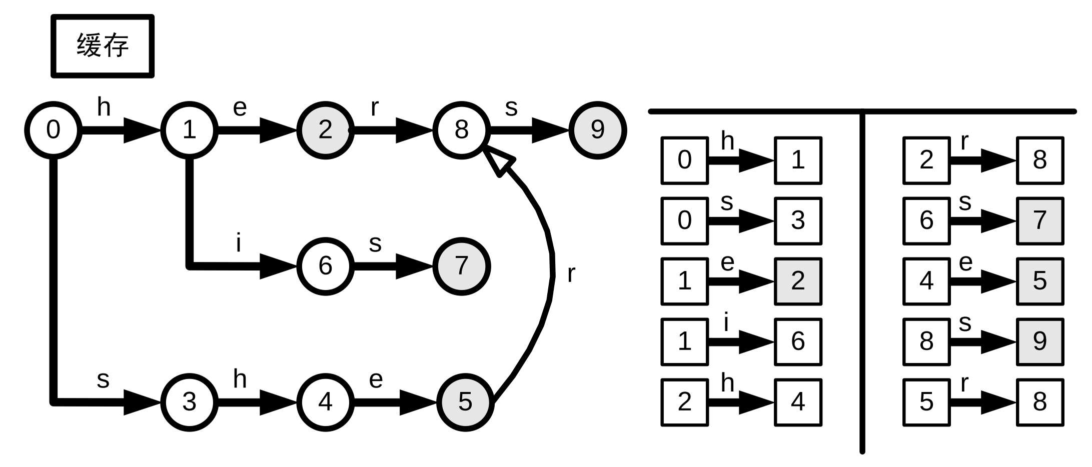



图4. 字符串集 $\{\boldsymbol{he},\boldsymbol{she},\boldsymbol{his},\boldsymbol{hers}\}$ 的 CDFA

宋天（音译，T. Song）等人[23]提出了一种基于 CDFA 的阿霍-克拉斯科算法，即采用缓存器记录额外状态来扩展确定自动机，减少冗余交叉迁移边，并采用下一状态寻址方法来高效存储和查找迁移边。图 4 给出了字符串集 $\{\boldsymbol{he},\boldsymbol{she},\boldsymbol{his},\boldsymbol{hers}\}$ 的 CDFA，其中左图是状态迁移，右图是对应的迁移边。与图 3 的 B-FSM 相比，图 4 的 CDFA 减少了部分交叉迁移边，即迁移到树深度为 2 的目的状态，而保留其他交叉迁移边；图 4 中的缓存是用于存储 1 个额外的当前状态。

CDFA 的匹配过程是：设置初始当前状态 ***current_state*** 为 0，缓存中额外状态 ***cache_state*** 为初始状态 0；当读入一个字符时，从状态迁移表中查找 ***current_state*** 和

***cache_state*** 对应的下一状态分别为 ***next_state*1** 和 ***next_state*2**，如果 ***next_state*1** 存在，则设置 ***current_state*** 为 ***next_state*1**，并查找初始状态 0 对应的下一状态 ***next_state*0**，设置 ***cache_state*** 为 ***next_state*0**；如果 ***next_state*1** 不存在且 ***next_state*2** 存在，则设置 ***current_state*** 为 ***next_state*2** 且 ***cache_state*** 为 ***next_state*0**；如果 ***next_state*1** 和 ***next_state*2** 均不存在，设置 ***cache_state*** 为 ***next_state*0** 而 ***cache_state*** 为 0。

例如，在图 4 中，当读入一个字符串 {***shishe***} 时，在 CDFA 中查找所有匹配字符串；当读入一个字符 ***s*** 时，设置 ***current_state*** 和 ***cache_state*** 均为 0，***next_state*1** 和 ***next_state*2** 均为 3，则迁移到下一状态 3，设置 ***current_state*** 和 ***cache_state*** 均为 3；当读入下一个字符 ***h*** 时，***next_state*1** 为 4 且 ***next_state*2** 为 4，则迁移到下一状态 4，设置 ***current_state*** 为 4，并查找出 ***next_state*0** 为 1，设置 ***cache_state*** 为 1；读入下一个字符 ***i*** 时，***next_state*1** 不存在且 ***next_state*2** 为 6，则设置 ***current_state*** 为 6，并查找出 ***next_state*0** 为 0，设置 ***cache_state*** 为 0；当读入下一个字符 ***s*** 时，***next_state*1** 为 7 且 ***next_state*2** 为 3，则迁移到状态 7，设置 ***current_state*** 为 7，并查找出 ***next_state*0** 为 3，设置 ***cache_state*** 为 3，输出匹配字符串集 {***his***}；当读入下一个字符 ***h*** 时，***next_state*1** 不存在且 ***next_state*2** 为 4，则设置 ***current_state*** 为 4，并查找出 ***next_state*0** 为 1，设置 ***cache_state*** 为 1；当读入下一个字符 ***e*** 时，***next_state*1** 为 5 且 ***next_state*2** 为 2，则设置 ***current_state*** 为 5，并查找出 ***next_state*0** 为 0，设置 ***cache_state*** 为 0，输出匹配字符串集 {***she,he***}。

### 3.3 多字符字符串匹配算法

为了提高字符串匹配算法的吞吐量，研究者提出了多字符字符串匹配算法，包括固定步长和变步长的多字符字符串匹配算法。固定步长是指每次读入相同大小的多个字符来进行确定自动机状态迁移；而变步长是指每次读入不同大小的多个字符来进行确定自动机状态迁移。本节分别介绍固定步长和变步长的阿霍-克拉斯科算法。

#### 3.3.1 固定步长的多字符阿霍-克拉斯科算法

达马普利卡（S. Dharmapurikar）等人[24]提出了一种基于 JACK-NFA (Jump-ahead Aho-Corasick NFA) 的阿霍-克拉斯科算法，即在确定自动机基础上，通过跳跃固定步长的多个字符来构建一个 JACK-NFA，并采用布隆过滤器来存储和查找迁移边。图 5 给出了字符串集 {***abcd, cde, bade, bc***} 的 JACK-NFA，其固定步长为 2。如图 5 所示，左图是状态迁移，右图是对应的迁移边；根据匹配字符长度，迁移边可分为 3 组，并采用布隆过滤器来表示每组迁移边。状态迁移表的每个元素包含 4 个字段，即<源状态，匹配字符>、目的状态、额外状态和匹配字符串。

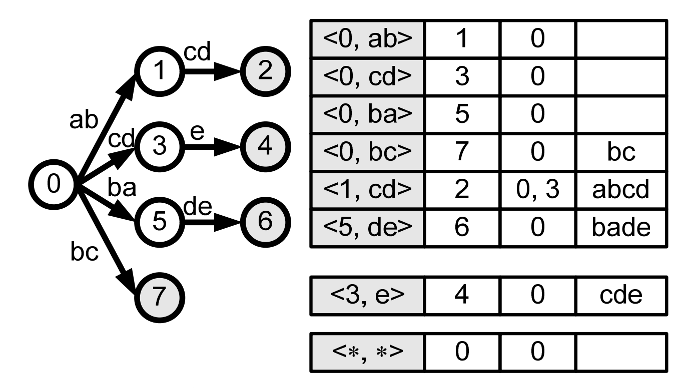
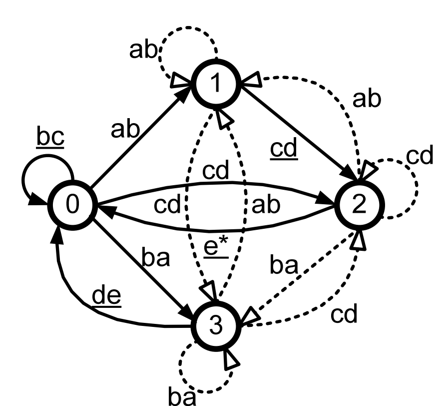



图5. 字符串集 {***abcd,cde,bade,bc***} 的 JACK-NFA

JACK-NFA 的匹配过程是：设置初始当前状态为 0，采用 DFA1 查找一个读入字符串，采用 DFA2 查找一个偏移 1 个字符的读入字符串，且 DFA1 和 DFA2 并行执行；当读入一个字符时，查找状态迁移表对应的布隆过滤器，每个确定自动机的当前状态迁移到下一状态；如果下一个状态为匹配状态，输出所有匹配字符串。例如，在图 5 中，读入一个字符串 {***abadeabcdea***}，在 JACK-NFA 中查找所有匹配字符串；设置初始状态 0 为当前状态，DFA1

读入一个字符串{***abadeabcdea***}，而 DFA2 读入一个字符串{***badeabcdea***}；DFA1 执行状态迁移为 $0 \rightarrow \{1,0\} \rightarrow \{0,0\} \rightarrow \{0,0\} \rightarrow \{7,0\} \rightarrow \{0,0\} \rightarrow \{0,0\}$，输出匹配字符串集{***bc***}；DFA2 执行状态迁移为 $0 \rightarrow \{0,0\} \rightarrow \{5,0\} \rightarrow \{6,0\} \rightarrow \{1,0\} \rightarrow \{2,0,3\} \rightarrow \{4,0\}$，输出匹配字符串集{***bade***,***abcd***,***cde***}。

卢宏斌（音译，Hongbin Lu）等人[25]提出了一种基于 TDP-DFA 的阿霍-克拉斯科算法，即通过跳跃相同大小的多个字符来构建一个 TDP-DFA，并采用 BCAM[9]存储和并行查找迁移边。图 6 给出了字符串集{***abcd***,***cde***,***bade***,***bc***}的 TDP-DFA，其固定步长为 2。如图 6 所示，左图是状态迁移，右图是对应的 1 迁移边；以源状态和匹配字符为搜索关键值，采用 BCAM 来表示和查找迁移边。状态迁移表的每个元素包含 3 个字段，即<源状态，匹配字符>、目的状态和匹配标记，其中匹配字符串存储在片外存储器中。




| | | |
|---|---|---|
| <0, ab> | 1 | 0 |
| <0, cd> | 2 | 0 |
| <0, ba> | 3 | 0 |
| <0, bc> | 0 | 1 |
| <1, cd> | 2 | 1 |
| <1, ab> | 1 | 0 |
| <1, ba> | 3 | 0 |
| <2, e*> | 0 | 1 |

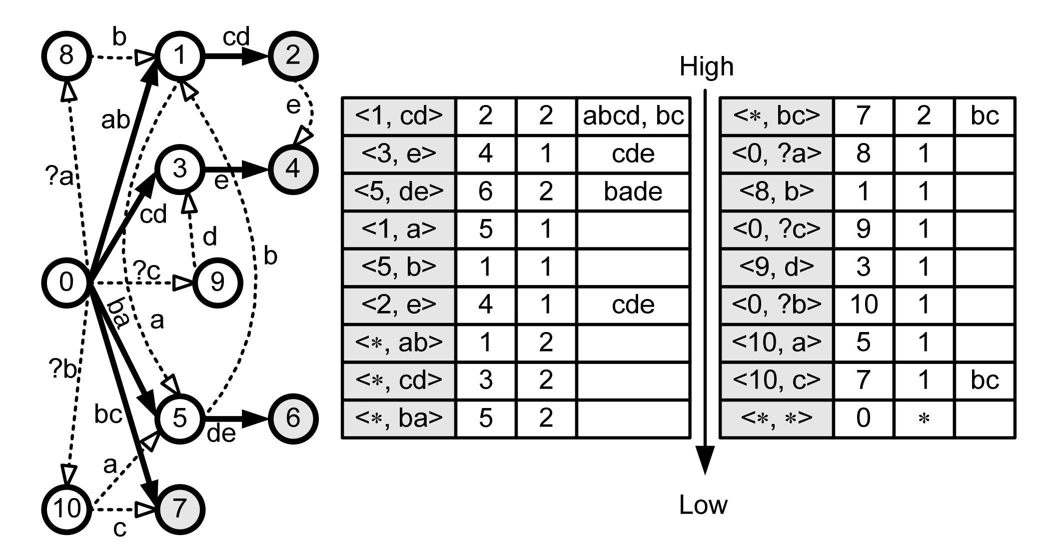
| | | |
|---|---|---|
| <2, ba> | 3 | 0 |
| <2, cd> | 2 | 0 |
| <2, ab> | 1 | 0 |
| <3, de> | 0 | 1 |
| <3, ab> | 1 | 0 |
| <3, ba> | 3 | 0 |
| <3, cd> | 2 | 0 |
| <*, *> | 0 | 0 |

图6. 字符串集{***abcd***,***cde***,***bade***,***bc***}的 TDP-DFA

与 JACK-NFA 相类似，TDP-DFA 的匹配过程是：设置初始当前状态为 0，采用确定自动机 DFA1 查找一个读入字符串，采用确定自动机 DFA2 查找一个偏移 1 个字符的读入字符串，且 DFA1 和 DFA2 是并行执行；当读入一个字符时，查找状态迁移表的 BCAM，每个确定自动机从当前状态迁移到下一状态；如果下一个状态为匹配状态，输出所有的匹配字符串。例如，在图 6 中，读入一个字符串{***abadeabcdea***}，在 TDP-DFA 中查找所有匹配字符串；从初始状态 0 开始，DFA1 读入一个字符串{***abadeabcdea***}，而 DFA2 读入一个字符串{***badeabcdea***}；DFA1 执行状态迁移为 $0 \rightarrow 1 \rightarrow 0 \rightarrow 0 \rightarrow 0 \rightarrow 0 \rightarrow 0$，输出匹配字符串集{***bc***}；DFA2 执行状态迁移为 $0 \rightarrow 0 \rightarrow 3 \rightarrow 0 \rightarrow 1 \rightarrow 2 \rightarrow 0$，输出匹配字符串集{***bade***,***abcd***,***cde***}。

艾利且瑞（M. Alicherry）等人[26]提出了一种基于压缩确定自动机的阿霍-克拉斯科算法，即在原始确定自动机基础上构建一个处理多字符的压缩确定自动机，支持回滚(Rollback)查找和最长输入匹配等操作，并采用 TCAM 实现迁移边的快速查找。图 7 给出了字符串集{***abcd***,***cde***,***bade***,***bc***}的压缩确定自动机，其固定步长为 2。如图 7 所示，左图是状态迁移，右图是对应的迁移边；状态迁移表的每个元素包含 4 个字段，即<源状态，匹配字符>、目的状态、偏移步长和匹配字符串；表中状态迁移边是按照优先级从高到低、从左到右的次序排列，即基本迁移边的优先级最高，重启迁移边的优先级次之，其他拆分迁移边又次之，失效迁移边的优先级最低。

[9] Binary Content Addressable Memory，双态内容可寻址存储器




| | | | |
|---|---|---|---|
| <1, cd> | 2 | 2 | abcd, bc |
| <3, e> | 4 | 1 | cde |
| <5, de> | 6 | 2 | bade |
| <1, a> | 5 | 1 | |
| <5, b> | 1 | 1 | |
| <2, e> | 4 | 1 | cde |
| <*, ab> | 1 | 2 | |
| <*, cd> | 3 | 2 | |
| <*, ba> | 5 | 2 | |

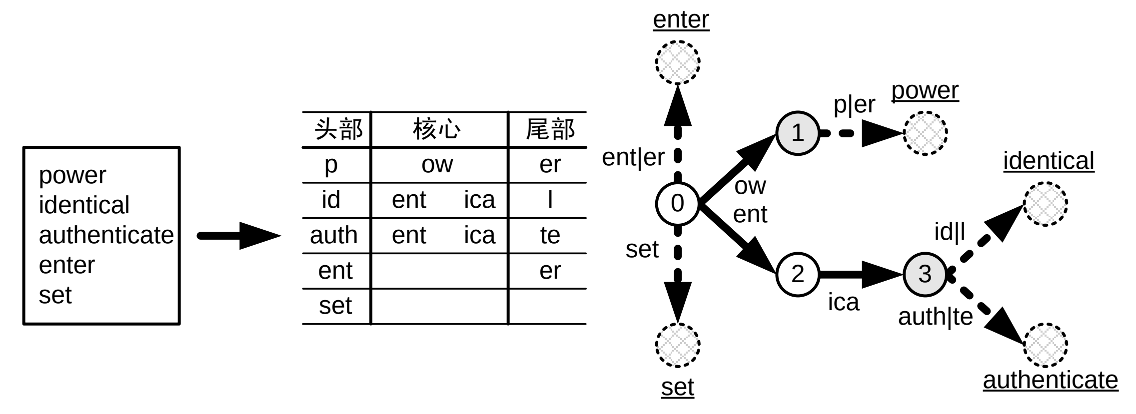
| | | | |
|---|---|---|---|
| <*, bc> | 7 | 2 | bc |
| <0, ?a> | 8 | 1 | |
| <8, b> | 1 | 1 | |
| <0, ?c> | 9 | 1 | |
| <9, d> | 3 | 1 | |
| <0, ?b> | 10 | 1 | |
| <10, a> | 5 | 1 | |
| <10, c> | 7 | 1 | bc |
| <*, *> | 0 | * | |

图7. 字符串集 $\{\boldsymbol{abcd},\boldsymbol{cde},\boldsymbol{bade},\boldsymbol{bc}\}$ 的压缩确定自动机

压缩确定自动机的匹配过程是：设置初始当前状态为 0，当读入一个字符时，查找状态迁移表对应的 TCAM，当前状态迁移到下一状态；如果下一个状态为匹配状态，输出所有匹配字符串。例如，在图 7 中，读入一个字符串 $\{\boldsymbol{abadeabcdea}\}$，在压缩确定自动机上查找所有匹配字符串；从初始状态 0 开始，根据匹配字符的不同步长，压缩确定自动机执行状态迁移为 $0 \to \{\boldsymbol{ab}:1\} \to \{\boldsymbol{a}:5\} \to \{\boldsymbol{de}:6\} \to \{\boldsymbol{ab}:1\} \to \{\boldsymbol{cd}:2\} \to \{\boldsymbol{e}:4\} \to \{\boldsymbol{a}:8\}$，其中 $\{\}$ 表示匹配字符和目的状态，输出匹配字符串集 $\{\boldsymbol{bade},\boldsymbol{abcd},\boldsymbol{bc},\boldsymbol{cde}\}$。

### 3.3.2 变步长的多字符阿霍-克拉斯科算法

华楠（音译，Nan Hua）等人[27]提出了一种基于变步长确定自动机的阿霍-克拉斯科算法，即采用风选(Winnowing)算法将特征字符串和读入字符串分割成不同大小的字符子串，构建一个变步长确定自动机，并采用哈希表查找核心迁移边以及采用 TCAM 查找短迁移边。图 8 给出了字符串集 $\{\boldsymbol{power},\boldsymbol{identical},\boldsymbol{authenticate},\boldsymbol{enter},\boldsymbol{set}\}$ 的变步长确定自动机，其中左图是将特征字符串分割成子串，右图是状态迁移。如图 8 所示，采用风选算法将特征字符串分割成 3 个子串，即头部子串、核心子串和尾部子串，其中核心子串对应的迁移边称为核心迁移边，头部子串和尾部子串对应的迁移边称为短迁移边。在图 8 的状态迁移中，虚线迁移边是短迁移边，实线迁移边是核心迁移边；短迁移边采用 TCAM 查找出匹配状态(即虚线节点)。



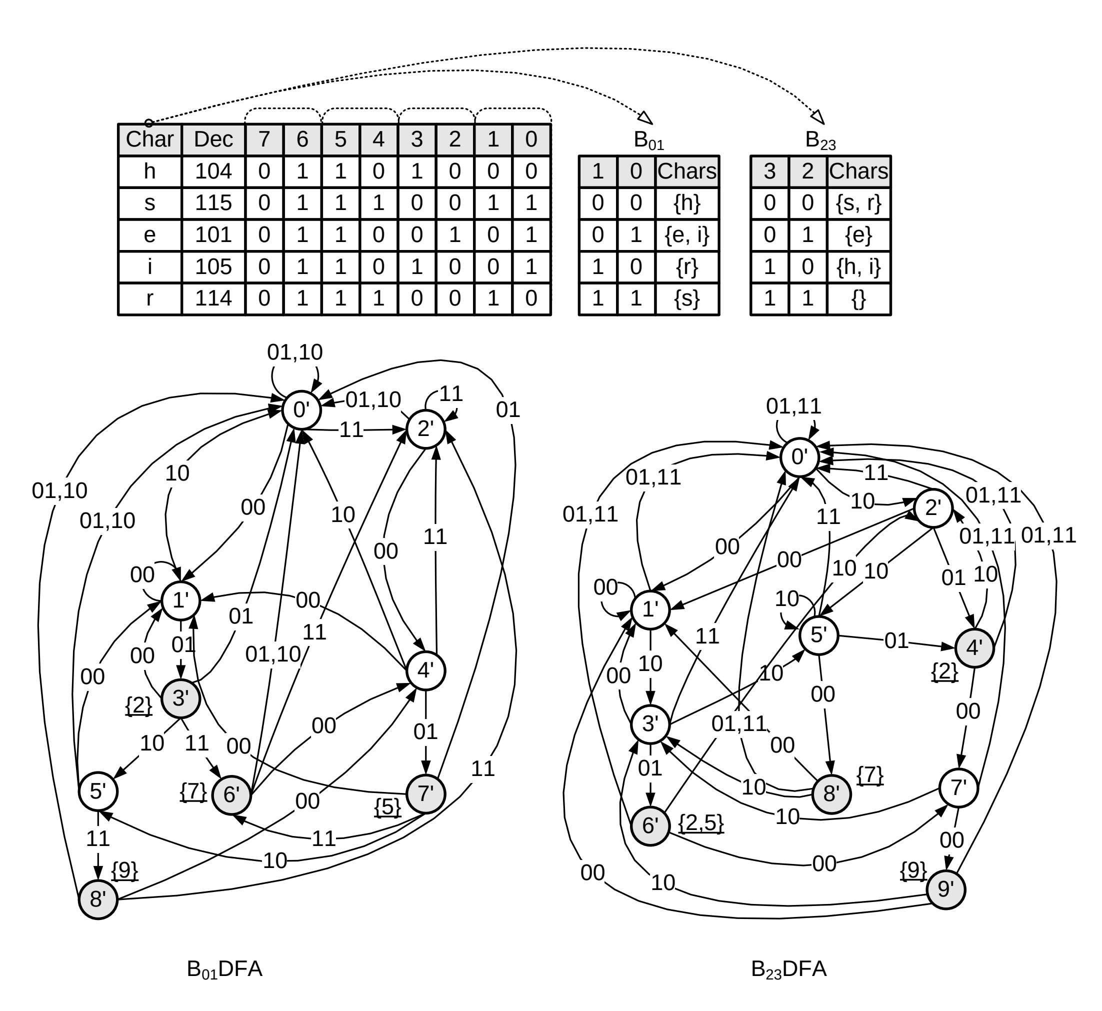

| 头部 | 核心 | | 尾部 |
|---|---|---|---|
| p | ow | | er |
| id | ent | ica | l |
| auth | ent | ica | te |
| ent | | | er |
| set | | | |

图8. 字符串集 $\{\boldsymbol{power},\boldsymbol{identical},\boldsymbol{authenticate},\boldsymbol{enter},\boldsymbol{set}\}$ 的变步长确定自动机

## 3.4 并行字符串匹配算法

为了实现线速 DPI，研究者提出了并行字符串匹配算法，即利用并行化提高确定自动机的匹配速率，压缩确定自动机存储空间需求，从而提高 DPI 的匹配吞吐量。本文主要介绍基于专用嵌入式硬件的比特拆分阿霍-克拉斯科算法、基于多核处理器平台的首尾分割阿霍-克拉斯科算法、基于并行布隆过滤器的阿霍-克拉斯科算法。

### 3.4.1 比特拆分阿霍-克拉斯科算法



| Char | Dec | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|
| h | 104 | 0 | 1 | 1 | 0 | 1 | 0 | 0 | 0 |
| s | 115 | 0 | 1 | 1 | 1 | 0 | 0 | 1 | 1 |
| e | 101 | 0 | 1 | 1 | 0 | 0 | 1 | 0 | 1 |
| i | 105 | 0 | 1 | 1 | 0 | 1 | 0 | 0 | 1 |
| r | 114 | 0 | 1 | 1 | 1 | 0 | 0 | 1 | 0 |

$B_{01}$

| 1 | 0 | Chars |
|---|---|---|
| 0 | 0 | {h} |
| 0 | 1 | {e, i} |
| 1 | 0 | {r} |
| 1 | 1 | {s} |

$B_{23}$

| 3 | 2 | Chars |
|---|---|---|
| 0 | 0 | {s, r} |
| 0 | 1 | {e} |
| 1 | 0 | {h, i} |
| 1 | 1 | {} |



图9. 字符串集 $\{\boldsymbol{he},\boldsymbol{she},\boldsymbol{his},\boldsymbol{hers}\}$ 的比特拆分确定自动机

谭琳（音译，Lin Tan）等人[28]提出了一种面向嵌入式硬件实现的比特拆分(Bit-Split) 阿霍-克拉斯科算法，即将原始确定自动机拆分为一组并行的微型确定自动机，且每个微型确定自动机并行处理每个读入字符的指定比特子串。图 9 给出了字符串集 $\{\boldsymbol{he},\boldsymbol{she},\boldsymbol{his},\boldsymbol{hers}\}$ 的比特拆分确定自动机，其中上图是将原始字母表拆分成微型字母表，即 $\Sigma\boldsymbol{B}_{01}$ 和 $\Sigma\boldsymbol{B}_{23}$，下图是对应的两个微型确定自动机，即 $\boldsymbol{B}_{01}\boldsymbol{DFA}$ 和 $\boldsymbol{B}_{23}\boldsymbol{DFA}$。如图 9 所示，在原始字母表 $\{\boldsymbol{e},\boldsymbol{h},\boldsymbol{i},\boldsymbol{r},\boldsymbol{s}\}$ 的比特拆分中，每个 8 位比特的字符被拆分为 4 个 2 位比特子串，从而构建相应的微型字母表。微型字母表 $\Sigma\boldsymbol{B}_{01}$ 的每个元组包括第 0 和 1 位比特子串及相应的字符子集。例如，第 2 个元组包含 2 位比特子串 01 和相应的字符子集 $\{\boldsymbol{e},\boldsymbol{i}\}$；微型字母表 $\Sigma\boldsymbol{B}_{23}$ 的每个元组包括第 2 和 3 位比特子串及相应的字符子集。

比特拆分确定自动机的匹配过程是：每个读入字符被拆分为一组 $\boldsymbol{b}$ 位比特子串，每个子串被分发给相应的微型确定自动机；每个微型确定自动机从初始微型状态 0′ 开始，并行迁移到下一个状态；对所有下一个微型状态对应的原始状态子集进行求交集运算，输出原始匹

配状态及其对应的特征字符串。例如，在图 9 中，当读入一个字符串 {***rhe***} 时，$B_{01}DFA$ 读入的 2 位比特子串集为 {10,00,01}，而 $B_{23}DFA$ 读入的 2 位比特子串集为 {10,10,01}；$B_{01}DFA$ 执行状态迁移为 $0' \to 1' \to 3' \to 6'$；$B_{23}DFA$ 也执行状态迁移为 $0' \to 1' \to 3' \to 6'$；由于 $B_{01}DFA$ 的状态 $0'$ 和 $1'$ 不是匹配状态，且 $B_{23}DFA$ 的状态 $0'$、$1'$ 和 $3'$ 也不是匹配状态，对 $B_{01}DFA$ 的状态 $3'$ 和 $B_{23}DFA$ 的状态 $6'$ 进行求交集运算，输出原始匹配状态 {2} 及其对应的匹配字符串集 {***he***}。

### 3.4.2 首尾分割阿霍-克拉斯科算法

杨怡华（音译，Yi-Hua Yang）等人[29]提出了一种基于首尾分割确定自动机的阿霍-克拉斯科算法，即将原始确定自动机分割成 1 个首部确定自动机和多个并行尾部非确定型有限自动机，并利用多核处理器的多核多线程来实现首尾分割确定自动机。图 10 给出了字符串集 {***he***,***she***,***his***,***hers***} 的首尾分割确定自动机，即沿着树深度为 2 进行分割，其中左图是 1 个首部确定自动机，右图是 3 个并行尾部非确定型有限自动机。如图 10 所示，状态 2、6 和 4 是触发状态，即当首部确定自动机迁移到触发状态时，启动相应的尾部非确定型有限自动机线程进行状态迁移。在多核处理器平台上，首部确定自动机是主线程，尾部非确定型有限自动机是子线程，当主线程启动子线程后，主线程与子线程并行执行。

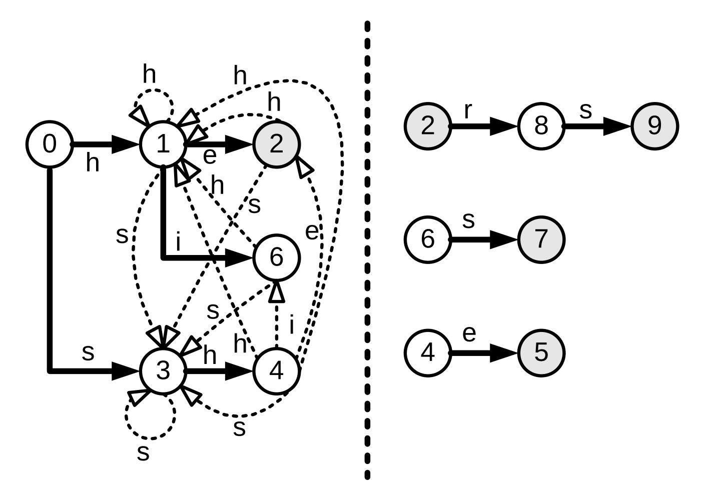



图10. 字符串集 {***he***,***she***,***his***,***hers***} 的首尾分割确定自动机

首尾分割确定自动机的匹配过程是：设置初始当前状态为 0，当读入一个字符时，首部确定自动机的当前状态迁移到下一个状态；如果下一状态为触发状态，触发相应的尾部非确定型有限自动机线程，设置触发状态为尾部非确定型有限自动机的当前状态，迁移到相应的下一状态；并设置触发状态为首部确定自动机的当前状态，并依次执行状态迁移；如果首部确定自动机或尾部非确定型有限自动机的下一个状态为匹配状态，输出所有匹配字符串。例如，在图 10 中，当读入一个字符串 {***sohershe***} 时，从初始状态 0 开始，首部确定自动机执行状态迁移为 $0 \to 3 \to 0 \to 1 \to 2 \to 0 \to 3 \to 4 \to 2$，当迁移到状态 2 或 4 时，先后启动 2 次尾部非确定型有限自动机线程分别执行状态迁移为 $2 \to 8 \to 9$ 和 $4 \to 5$，并输出匹配字符串集 {***he***,***she***,***his***,***hers***}。

### 3.4.3 基于并行布隆过滤器的阿霍-克拉斯科算法

达马普利卡等人[30]提出了一种基于并行布隆过滤器的字符串匹配算法，即依据特征字符串长度不同，将特征字符串集分成多个字符串子集，采用布隆过滤器[32]来表示每个子集，且多个布隆过滤器是并行查找。图 11 给出了基于并行布隆过滤器的字符串匹配体系架构，其中每个布隆过滤器表示一组长度相同的特征字符串。

并行布隆过滤器的匹配过程是：读入一个字符串，从串头开始，依次选择多个长度累加的子串，并行查找对应的布隆过滤器，输出所有查找成功的读入字符子串。例如，在图 11 中，$W-2$个并行布隆过滤器分别对长度为 $3,\cdots W$ 的读入字符子串进行并行查找。由于布隆过滤器会产生假阳性错误[32]，即不属于集合 $S$ 的元素被误判为属于该集合，基于并行布隆过滤器的字符串匹配算法也会产生假阳性匹配字符串，因而需要查找片外哈希表进行验证。该算法要求高吞吐量的并行存储器访问带宽，存在代价昂贵和可伸缩性差等问题。

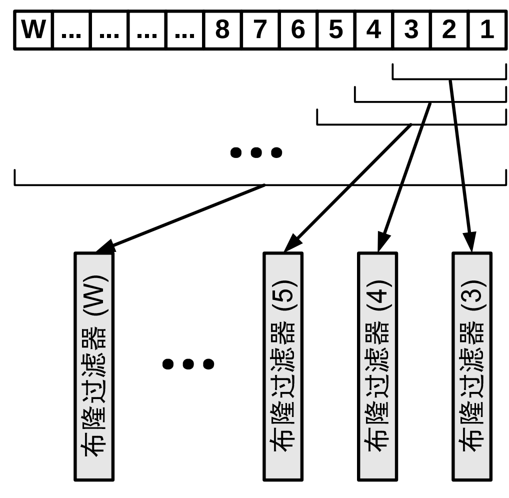



图11. 基于并行布隆过滤器的字符串匹配体系架构

## 4 基于硬件的正则表达式匹配算法

随着智能攻击，例如躲避攻击[33]、多态或变型攻击[34-35]等不断涌现，字符串难以准确描述这些攻击的复杂特征，导致字符串匹配算法的检测率降低。由于正则表达式具有丰富和灵活的表达能力[36]，当前的 NIDS/NIPS 已采用正则表达式替代字符串来描述攻击特征，并实现正则表达式匹配算法。

表 2 基于硬件的正则表达式匹配算法分类

| | 迁移边压缩算法 | 状态压缩算法 |
|---|---|---|
| $D^2FA$[10] | √ | |
| $CD^2FA$[11] | √ | |
| 状态融合 DFA | | √ |
| XFA[12] | | √ |

基于硬件的正则表达式匹配算法可分为：迁移边压缩的正则表达式匹配算法和状态压缩的正则表达式匹配算法。如表 2 所示，迁移边压缩算法是研究如何通过减少冗余迁移边来压缩确定自动机存储空间开销，例如基于 $D^2FA$[37]和基于 $CD^2FA$[38]的正则表达式匹配算法；状态压缩算法是研究如何通过减少冗余状态来压缩确定自动机存储空间开销，例如基于状态融合的确定型有限自动机[39]和基于扩展有限自动机[40-41]的正则表达式匹配算法。本节分别介绍原始正则表达式匹配算法、迁移边压缩和状态压缩的正则表达式匹配算法。

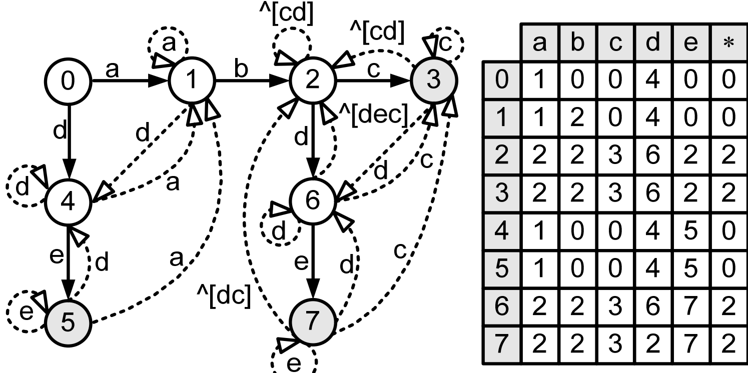
### 4.1 原始正则表达式匹配算法

正则表示式匹配算法的原始确定自动机构建过程分为模式编译和子集构造[37]。与阿霍-克拉斯科算法相类似，模式编译过程是构建一颗从初始状态到匹配状态的有向特里树；子集构造过程是从初始状态开始，为每个状态构建确定性迁移边，即将非确定型有限自动机等价转换为确定型有限自动机。正则表达式具有灵活和丰富的特殊字符，例如“+”表示任意一个以上字符，“*”表示任意多个字符，“｜”表示或关系，“·”表示单个字符通配符，“？”表示任意一个字符，“{}”表示字符重复，“[ ]”表示字符子集，“[^]”表示非字符子集。图 12 给出了

[10] Delayed Input DFA，延迟输入的确定型有限自动机
[11] Content Addressed Delayed Input DFA，内容寻址的延迟输入的确定型有限自动机
[12] Extended Finite Automaton，扩展有限自动机

正则表达式集{·*ab·*c,·*de}的原始确定自动机，其中左图是状态迁移，右图是对应的状态迁移表。如图 12 所示，在状态迁移表中，列项“*”表示除字母表{*a*,*b*,*c*,*d*,*e*}之外的其他字符，用于匹配失效迁移边。

原始确定自动机的匹配过程是：设置初始当前状态为 0，当读入一个字符时，当前状态迁移到下一个状态，并设置下一个状态为当前状态，依次执行状态迁移；如果下一个状态为匹配状态，输出所有匹配字符串。例如，在图 12 中，当读入一个字符串{*abadcade*}时，从初始状态 0 开始，原始确定自动机执行状态迁移为 $0\rightarrow1\rightarrow2\rightarrow2\rightarrow6\rightarrow3\rightarrow2\rightarrow6\rightarrow7$，并输出匹配字符串集{·*ab*·*c*,·*de*}。




| | a | b | c | d | e | * |
|---|---|---|---|---|---|---|
| 0 | 1 | 0 | 0 | 4 | 0 | 0 |
| 1 | 1 | 2 | 0 | 4 | 0 | 0 |
| 2 | 2 | 2 | 3 | 6 | 2 | 2 |
| 3 | 2 | 2 | 3 | 6 | 2 | 2 |
| 4 | 1 | 0 | 0 | 4 | 5 | 0 |
| 5 | 1 | 0 | 0 | 4 | 5 | 0 |
| 6 | 2 | 2 | 3 | 6 | 7 | 2 |
| 7 | 2 | 2 | 3 | 2 | 7 | 2 |

图12.　正则表达式集{·*ab*·*c*,·*de*}的原始确定自动机

确定自动机存储空间大小是由状态个数和每个状态的迁移边条数所决定的。对于正则表达式匹配算法，造成确定自动机存储空间需求大的主要原因是：

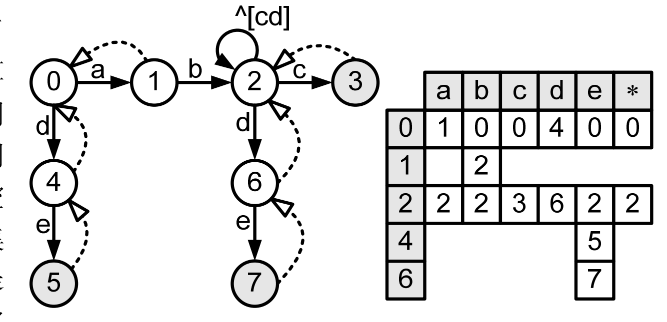
- 在迁移边方面，确定自动机存在许多冗余迁移边，包括状态间和状态内的冗余迁移边；
- 在状态方面，由于正则表达式采用许多语义丰富的独特符号，例如“*”，“{}”和“[ ]”等，确定自动机需要大量的额外状态来记录部分匹配结果，导致其状态空间爆炸。

因此，研究者主要从状态和迁移边等两方面压缩确定自动机存储空间需求。

## 4.2 迁移边压缩的正则表达式匹配算法

### 4.2.1 基于 $D^2FA$ 的正则表达式匹配算法

库玛（S. Kumar）等人[37]提出了一种基于 $D^2FA$ 的正则表达式匹配算法，即采用一条默认迁移边替代不同状态之间的相同迁移边，而保留不同迁移边，从而减少确定自动机存储空间开销。图 13 给出了正则表达式集{·*ab*·*c*,·*de*}的 $D^2FA$，其中左图是状态迁移，右图是对应的非默认迁移边。图中，在 $D^2FA$，虚线表示默认迁移边，实线表示正常迁移边。例如，当读入一个字符 ***a*** 时，当前状态 5 查找其默认迁移边，迁移到状态 4；由于不存在匹配字符 ***a*** 的正常迁移边，状态 4 又查找其默认迁移边，迁移到状态 0；状态 0 查找匹配字符 ***a*** 的正常迁移边，最后迁移到状态 1。因此，状态 5 执行状态迁移为 $\boldsymbol{a}:5\rightarrow\{4,0,1\}$。




| | a | b | c | d | e | * |
|---|---|---|---|---|---|---|
| 0 | 1 | 0 | 0 | 4 | 0 | 0 |
| 1 | | 2 | | | | |
| 2 | 2 | 2 | 3 | 6 | 2 | 2 |
| 4 | | | | | 5 | |
| 6 | | | | | 7 | |

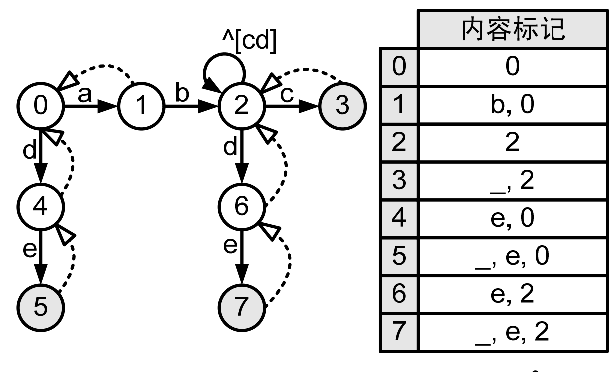
图13.　正则表达式集{·*ab*·*c*,·*de*}的 $D^2FA$

$D^2FA$ 的匹配过程是：设置初始当前状态为 0，当读入每个字符时，当前状态查找其正

常迁移边是否匹配成功。如果匹配成功，当前状态迁移到下一个状态，并设置下一个状态为当前状态，继续匹配下一个字符；如果匹配不成功，当前状态查找其默认迁移边，迁移到下一个状态，该状态查找其正常迁移边是否匹配成功；如果匹配成功，该状态迁移到下一个状态，否则，继续查找其默认迁移边，直到其正常迁移边匹配成功。例如，在图 13 中，当读入一个字符串 $\{\boldsymbol{abadcade}\}$ 时，在 D$^2$FA 中查找出所有匹配字符串；从初始状态 0 开始，D$^2$FA 执行状态迁移为 $0\to1\to2\to2\to6\to\{2,3\}\to\{2,2\}\to6\to7$，并输出匹配字符串集 $\{\cdot*\boldsymbol{ab}\cdot*\boldsymbol{c},\cdot*\boldsymbol{de}\}$。由于采用默认迁移边，D$^2$FA 虽然可显著减少存储空间开销，但是存在匹配吞吐量低等问题。例如，当读入一个字符串 $\{\boldsymbol{abadcade}\}$ 时，图 12 的原始确定自动机执行 8 次状态迁移，而图 13 的 D$^2$FA 执行 10 次状态迁移。

#### 4.2.2 基于 CD$^2$FA 的正则表达式匹配算法

库玛等人[38]提出了一种基于 CD$^2$FA 的正则表达式匹配算法，即在 D$^2$FA 的每个状态上增加下一默认迁移边的部分内容标识，以消耗更多存储空间为代价来提高 D$^2$FA 的匹配吞吐量。图 14 给出了正则表达式集 $\{\cdot*\boldsymbol{ab}\cdot*\boldsymbol{c},\cdot*\boldsymbol{de}\}$ 的 CD$^2$FA，其中左图是状态迁移，右图是每个状态维护的额外内容标识。如图 14 所示，在 CD$^2$FA 中，每个状态的内容标识包括 2 类字段，即默认迁移边路径上每个状态的匹配字符和根节点。例如，状态 5 的内容标识为"$_,\boldsymbol{e},0$"，其中"$_$"表示状态 5 的空字符，"$\boldsymbol{e}$"表示状态 4 的有效字符，"0"表示默认迁移边路径的根节点 0。由于不包含默认迁移边，状态 0 和 2 的额外内容标识为自身状态。




| | 内容标记 |
|---|---|
| 0 | 0 |
| 1 | b, 0 |
| 2 | 2 |
| 3 | _, 2 |
| 4 | e, 0 |
| 5 | _, e, 0 |
| 6 | e, 2 |
| 7 | _, e, 2 |

图14. 正则表达式集 $\{\cdot*\boldsymbol{ab}\cdot*\boldsymbol{c},\cdot*\boldsymbol{de}\}$ 的 CD$^2$FA

与 D$^2$FA 相类似，CD$^2$FA 的匹配过程是：设置初始当前状态为 0，当读入每个字符时，当前状态查找其正常迁移边是否匹配成功。如果匹配成功，当前状态迁移到下一个状态，并设置下一个状态为当前状态，继续匹配下一个字符；如果匹配不成功，当前状态利用其额外内容标识，跳跃式查找默认迁移边路径上的下一匹配状态，当前状态直接迁移到该状态，并设置下一个状态为当前状态，依次状态迁移；如果下一状态是匹配状态，输出所有匹配字符串。例如，在图 14 中，当读入一个字符串 $\{\boldsymbol{abadcade}\}$ 时，在 CD$^2$FA 中查找出所有匹配字符串；从初始状态 0 开始，CD$^2$FA 执行状态迁移为 $0\to1\to2\to2\to6\to3\to2\to6\to7$，并输出匹配字符串集 $\{\cdot*\boldsymbol{ab}\cdot*\boldsymbol{c},\cdot*\boldsymbol{de}\}$。因此，与原始确定自动机相比，CD$^2$FA 执行相同次数的状态迁移，但是显著减少其确定自动机存储空间开销。

### 4.3 状态压缩的正则表达式匹配算法

#### 4.3.1 基于状态融合的正则表达式匹配算法

贝奇（M. Becchi）等人[39]提出了一种基于状态融合的确定自动机的正则表达式匹配算法，即采用迁移边标记方法来融合多个非等价状态，从而减少确定自动机存储空间开销，并确保其最坏情况匹配性能。图 15 给出了正则表达式集 $\{(\boldsymbol{abc}|\boldsymbol{def})\boldsymbol{g}+1\}$ 的状态融合确定自动机。其中左图是原始确定自动机，右图是对应的状态融合的确定自动机。如图 15 所示，状态融合确定自动机标记迁移边为"$\cdot$"和"$/$"，则"$\cdot0$"和"$\cdot1$"表示当执行状态迁移时，分别设置标记位为 0 和 1；"$/0$"和"$/1$"表示当标记位设置为 0 和 1 时才执行状态迁移。例如，当状态 3 和 4 融合成一个状态 $3-4$ 时，迁移边"$\boldsymbol{b}\cdot0/0$"表示当标记位为 0 时，执行状态迁移，

设置标记位为 0；迁移边“$c/0$”表示当标记为 0 时，执行状态迁移。

状态融合确定自动机的匹配过程是：设置初始当前状态为 0，当读入每个字符时，当前状态查找出匹配迁移边，检查该迁移边的标识位来决定是否执行状态迁移，或从当前状态迁移到下一个状态，设置相应的标识位以及下一个状态为当前状态，并依次状态迁移；如果下一个状态为匹配状态，输出所有匹配字符串。例如，在图 15 中，当读入一个字符串 $\{abcdefg\}$ 时，在状态融合确定自动机中查找所有匹配字符串；从初始状态 0 开始，状态融合确定自动机执行状态迁移为 $0\to\{1-2\}\to\{3-4\}\to5\to\{1-2\}\to\{3-4\}\to5\to6$，并输出匹配字符串集 $\{(abc|def)g+1\}$。

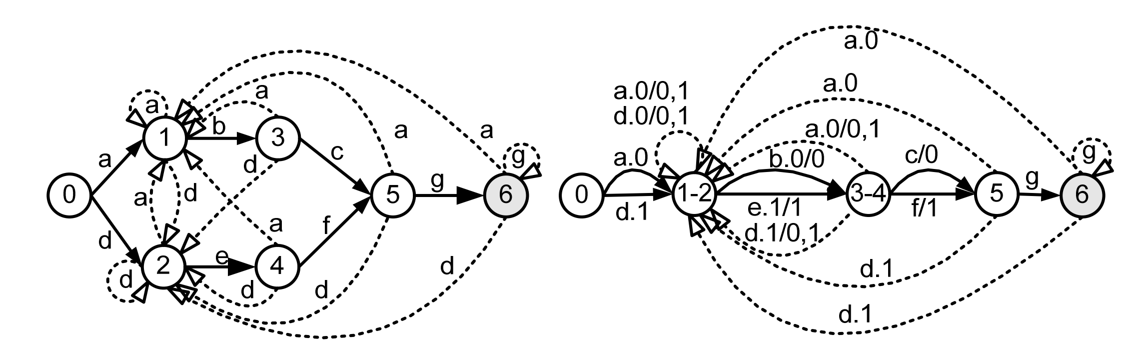



图15.　正则表达式集 $\{(abc|def)g+1\}$ 的状态融合确定自动机

#### 4.3.2 基于扩展有限自动机的正则表达式匹配算法

史密斯（R. Smith）等人[40-41]提出了一种基于扩展有限自动机的正则表达式匹配算法，即在状态上增加辅助变量和简单操作指令，避免确定自动机状态空间爆炸问题，从而减少确定自动机存储空间开销。对于正则表达式，确定自动机状态空间爆炸产生的根本原因是：当多个单独确定自动机合成一个组合确定自动机时，一个单独确定自动机的非歧义状态与其他单独确定自动机的歧义状态进行排列组合，导致组合确定自动机产生大量的额外状态，以记录所有部分匹配结果。扩展有限自动机采用辅助变量替代额外状态来记录部分匹配结果，执行简单操作指令来检查匹配是否成功。在扩展有限自动机中，操作指令主要包括赋值(Set)、重置(Reset)和比较(Compare)等。

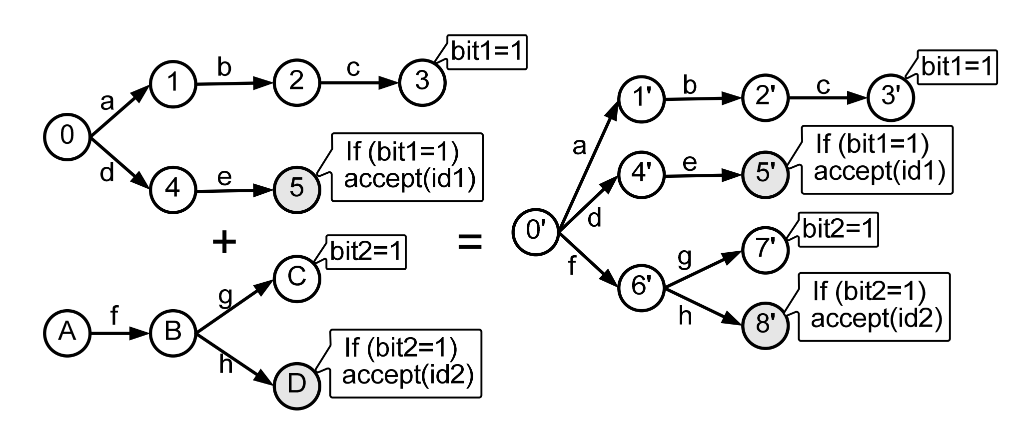



图16.　正则表达式集 $\{\cdot*abc\cdot*de,\cdot*fg\cdot*fh\}$ 的扩展有限自动机

图 16 给出了正则表达式集 $\{\cdot*abc\cdot*de,\cdot*fg\cdot*fh\}$ 的扩展有限自动机，其中左图是 2 个单独扩展有限自动机，右图是对应的组合扩展有限自动机（省略了交叉迁移边和失效迁移边)。如图 16 所示，对于正则表达式 $\{\cdot*abc\cdot*e\}$ 的单独扩展有限自动机，在状态 3 上增加了

赋值操作指令 $bit1=1$，表示当迁移到状态 3 时，设置 $bit1$ 为 1，用于记录已匹配 $abc$；在状态 5 上增加了比较操作指令 $if(bit1=1)$，表示当迁移到状态 5 时，检查 $bit1$ 是否为 1，如果 $bit1$ 为 1，则成功匹配 $id1$，输出匹配正则表达式 $\{.*ab.*cd\}$。相类似地，对于正则表达式 $\{.*fg.*fh\}$ 的单独扩展有限自动机，在状态 C 上增加了赋值操作指令 $bit2=1$，在状态 D 上增加了比较操作指令 $if(bit2=1)$。如图 16 所示，在组合扩展有限自动机的状态 $3'$ 和状态 $7'$ 上分别增加了 $bit1$ 和 $bit2$ 的赋值操作指令，在状态 $5'$ 和 $8'$ 上分别增加了 $bit1$ 和 $bit2$ 的比较操作指令。因此，当表示 $\{.*abc.*de,.*fg.*fh\}$ 时，组合扩展有限自动机仅需要 9 个状态和 2 个辅助比特变量。

扩展有限自动机匹配过程是：设置初始当前状态为 0，当读入每个字符时，当前状态查找相应的迁移边，迁移到下一个状态；执行下一状态的操作指令，检查辅助变量是否设置来判断匹配是否成功。例如，在图 16 中，当读入一个字符串 $\{abcfhde\}$ 时，在扩展有限自动机中查找出所有匹配字符串；从初始状态 $0'$ 开始，扩展有限自动机执行状态迁移为 $0' \to 1' \to 2' \to 3' \to 6' \to 8' \to 4' \to 5'$，并输出匹配字符串集 $\{.*abc.*de\}$。

# 5 结论与展望

DPI 技术面临高性能挑战，即如何满足高速数据包处理的存储空间需求和吞吐量，提升特征匹配的可伸缩性。本文是从时间和空间开销等方面综述了基于硬件的字符串匹配算法和正则表达式匹配算法，设计与实现时空高效数据包内容过滤。在基于硬件的字符串匹配算法中，本文主要介绍了面向硬件实现的存储高效字符串匹配算法、多字符字符串匹配算法以及并行字符串匹配算法；在基于硬件的正则表达式匹配算法中，本文主要介绍了迁移边压缩和状态压缩的正则表达式匹配算法。由于现代嵌入式硬件具有存储空间受限和并行计算能力强等特点，基于硬件的特征匹配算法研究目标是如何通过压缩确定自动机存储空间需求，在片上高速存储器上存储和查找整个确定自动机，从而实现高吞吐量的特征匹配。

随着 100Gbps 高速网络应用的日益普及，未来 DPI 技术将面临线速数据包处理的可伸缩性挑战，即 DPI 在处理速率、特征规则库、能耗和价格等方面的可扩展问题。特别是，高速存储器和多核处理器等硬件技术的日新月异，为可伸缩 DPI 技术带来新的机遇与挑战。未来 DPI 技术面临如下关键任务：

1. 基于多核处理器的并行 DPI 技术研究。多核处理器，例如 Intel Xeon CPU 和 GPGPU 集成了成百上千个 CPU 核，具有强大的并行计算能力。并行 DPI 技术研究面临如何细粒度并行化特征匹配算法[42]，在多个并行 CPU 核上实现特征匹配的数据并行、任务并行和流水线等多粒度并行；同时，进一步研究存储高效特征匹配算法，即在每个 CPU 核的高速缓冲器上存储和查找整个确定自动机，从而加速 DPI 的匹配吞吐量。
2. 基于 TCAM 的高效 DPI 技术研究。随着高速存储器技术的迅猛发展，TCAM 查找速率越来越快、存储空间越来越大，而其能耗和价格却不断降低。基于 TCAM 的 DPI 技术可应用于国家骨干网络的高速链路上，实现高速率大容量数据包识别与过滤。由于 TCAM 能耗与存储空间开销成正比，基于 TCAM 的高效 DPI 技术研究需要研究如何压缩确定自动机存储空间开销[43]，即从迁移边和状态等方面综合减少状态迁移表的存储空间需求，从而减少其能耗，提高 DPI 的匹配性能。
3. 面向网络虚拟化的高性能 DPI 技术研究。GENI 和 FIND 等下一代互联网研究项目采用网络虚拟化思想，即在一个互联网基础设施的硬件平台上动态运行多个相互隔离的虚拟网络及其应用[44]。支持虚拟化的可编程网络设备是实现网络虚拟化的关键部件，即在一个通用硬件平台上运行多个虚拟网络设备，例如虚拟化可编

程路由器和虚拟 NIDS/NIPS。高性能 DPI 技术研究面临如何在可编程网络设备的计算和存储能力受限条件下动态运行多个并行特征匹配算法，并确保每个特征匹配算法的存储空间需求和吞吐量，从而支持更多虚拟网络设备，提升其可伸缩性。

**参考文献：**


[1] S. Staniford, V. Paxson and N. Weaver, How to own the Internet in your spare time. In: Proceedings of USENIX Security Symposium, 2002.

[2] S. Singh, C. Estan and G. Varghese and S. Savage, Automated worm fingerprinting. In: Proceedings of OSDI, 2004.

[3] Frost and Sullivan. World intrusion detection and prevention systems markets. 2007.

[4] M. Roesch, Snort – lightweight intrusion detection for networks. In: Proceedings of LISA, 1999.

[5] V. Paxon, Bro: a system for detecting network intruders in real-time. Computer Networks, 1999, 31(23-24): 2435-2463

[6] Snort-the de facto standard for intrusion detection/prevention. http://www.snort.org, 2009.

[7] J. Levandoski, E. Sommer and M. Strait, Application layer packet classifier for Linux. http://l7-filter.sourceforge.net/, 2008.

[8] S. Sen, O. Spatscheck and D. Wang, Accurate, scalable in-network identification of P2P traffic using application signatures. In: Proceedings of WWW, 2004.

[9] D. E. Knuth, J. H. Morris and V. R. Pratt, Fast pattern matching in strings. SIAM Journal on Computing, 1977, 6(2): 323-350.

[10] R. S. Boyer and J. S. Moore, A fast string searching algorithm. Communications of the ACM, 1977, 20(10): 762-772.

[11] A. V. Aho and M. J. Corasick, Efficient string matching: an aid to bibliographic search. Communications of the ACM, 1975, 18(6): 333-340.

[12] B. Commentz-Walter, A string matching algorithm fast on the average. In: Proceedings of the 6th Colloquium on Automata, Languages and Programming, 1979.

[13] C. Coit, S. Staniford and J. McAlerney, Towards faster string matching for intrusion detection. In: Proceedings of DARPA Information Survivability Conference and Exhibition, 2002.

[14] R. Sidhu and V. K. Prasanna, Fast regular expression matching using FPGAs. In: Proceedings of IEEE FCCM, 2001.

[15] C. R. Clark and D. E. Schimmel, Efficient reconfigurable logic circuit for matching complex network intrusion detection patterns. In: Proceedings of IEEE FPL, 2003.

[16] J. Moscola, J. Lockwood, R. P. Loui and M. Pachos, Implementation of a content-scanning module for an internet firewall. In: Proceedings of IEEE FCCM, 2003.

[17] C. R. Clark and D. E. Schimmel, Scalable pattern matching on high-speed networks. In: Proceedings of IEEE FCCM, 2004.

[18] I. Sourdis and D. Pnevmatikatos, Pre-decoded CAMs for efficient and high-speed NIDS pattern matching. In: Proceedings of IEEE FCCM, 2004.

[19] Xilinx Vertex-5 family overview. http://www.xilinx.com, 2009.

[20] 黄昆，张大方，谢高岗，金军航，一种面向深度数据包检测的紧凑型正则表达式匹配算法. 中国科学: 信息科学, 2010, 40(2): 356-370.

[21] B. C. Brodie, R. K. Cytron and D. E. Taylor, A scalable architecture for high-throughput regular-expression pattern matching. In: Proceedings of ACM ISCA, 2006.

[22] J. van Lunteren, High performance pattern-matching for intrusion detection. In: Proceedings of IEEE INFOCOM, 2006.

[23] T. Song, W. Zhang and D. Wang, A memory efficient multiple pattern matching architecture for network security. In: Proceedings of IEEE INFOCOM, 2008.

[24] S. Dharmapurikar and J. Lockwood, Fast and scalable pattern matching for content filtering. In: Proceedings of ACM ANCS, 2005.

[25] H. Lu, K, Zheng, B. Liu, X. Zhang and Y. Liu, A memory-efficient parallel string matching architecture for high-speed intrusion detection. IEEE Journal on Selected Areas in Communication, 2006, 34(10): 1793-1804.

[26] M. Alicherry, M. Muthuprasanna and V. Kumar, High speed pattern matching for network IDS/IPS. In: Proceedings of IEEE ICNP, 2006

[27] N. Hua, H. Song and T. V. Lakshman, Variable-stride multi-pattern matching for scalable deep packet inspection. In: Proceedings of IEEE INFOCOM, 2009.

[28] L. Tan, B. Brotherton and T. Sherwood, Bit-split string-matching engines for intrusion detection and prevention. ACM Transactions on Architecture and Code Optimization, 2006, 3(1): 3-34.

[29] Y. E. Yang, V. K. Prasanna and C. Jiang, Head-body partitioned string matching for deep packet inspection with scalable and attack-resilient performance. In: Proceedings of IEEE IPDPS, 2010.

[30] S. Dharmapurikar, P. Krishnamurthy, T. S. Sproull and J W. Lockwood, Deep packet inspection using parallel bloom filters. IEEE Micro, 2004, 24(1): 52-61.

[31] N. Tuck, T. Sherwood, B. Calder and G. Verghese, Deterministic memory-efficient string matching algorithms for intrusion detection. In: Proceedings of IEEE INFOCOM, 2004.

[32] A. Broder and M. Mitzenmacher, Network applications of Bloom filters: a survey. Internet Mathematics, 2004, 1(4):485-509.

[33] M. Vutukuru, H. Balakrishna, and V. Paxson, Efficient and robust TCP stream normalization. In: Proceedings of IEE Symposium on Security and Privacy, 2008.

[34] C. Kanich, C. Kreibich, K. Levchenko, B. Enright, G. M. Voelker, V. Paxson and S. Savage, Spamalytics: an empirical analysis of spam marketing conversion. Communications of ACM, 2009, 52(9): 99-107.

[35] D. Brumley, J. Newsome, D. Song, H. Wang and S. Jha, Towards automatic generation of vulnerability-based signatures. In: Proceedings of IEE Symposium on Security and Privacy, 2006.

[36] R. Sommer and V. Paxson, Enhancing byte-level network intrusion detection signatures with context. In: Proceedings of ACM Computer and Communication Security, 2003.

[37] S. Kumar, S. Dharmapurikar, F. Yu, P. Crowley and J. Turner, Algorithms to accelerate multiple regular expressions matching for deep packet inspection. In: Proceedings of ACM SIGCOMM, 2006.

[38] S. Kumar, J. Turner and J. Williams, Advanced algorithms for fast and scalable deep packet inspection. In: Proceedings of ACM ANCS, 2006.

[39] M. Becchi and S. Cadambi, Memory-efficient regular expression search using state merging. In: Proceedings of IEEE INFOCOM, 2007.

[40] R. Smith, C. Estan and S. Jha, XFA: faster signature matching with extened automata. In: Proceedings of IEEE Symposium on Security and Privacy, 2008.

[41] R. Smith, C. Estan, S. Jha and S. Kong, Deflating the big bang: fast and scalable deep packet inspection with extened finite automat. In: Proceedings of ACM SIGCOMM, 2008.

[42] R. Sommer, V. Paxson and N. Weaver, An architecture for exploiting multi-core processors to parallelize network intrusion detection prevention. Concurrency and Computation: Practice and Experience, 2009, 21:1255-1279.

[43] C. R. Meiners, J. Patel, E. Norige, E. Torng and A. Liu,. Fast regular expression matching using small TCAMs for network intrusion detection and prevention systems. In: Proceedings of USENIX Security, 2010.

[44] A. Bavier, N. Feamster, M. Huang, L. Peterson and J. Rexford, In VINI veritas: realistic and controlled network experimentations. In: Proceedings of ACM SIGCOMM, 2006.

作者简介：

**黄 昆：** 中国科学院计算技术研究所网络技术研究中心，博士后 huangkun09@ict.ac.cn

**谢高岗：** 中国科学院计算技术研究所，网络技术研究中心主任，研究员，博士生导师